

### Instructions

Form a small group. Start on problem 1.1. Check off with a staff member or discuss your *solution process* with a nearby group when you think everyone in your group understands *how to solve* problem 1.1. Then repeat for 1.2, 1.3, ...

You may not move to the next problem until you check off or discuss with another group and *everyone understands why the solution is what it is*. You may use any course resources at your disposal: the purpose of this review session is to have everyone learning together as a group.

## 1 Functions

1.1 Explain the difference between the following:

(a) 

```
>>> def square(x):  
...     return x * x
```

(b) 

```
>>> square(4)
```

(c) 

```
>>> square
```

1.2 What would Python display?

(a) 

```
(lambda x: x(x))(lambda y: 4)
```

(b) 

```
(lambda x, y: y(x))(mul, lambda a: a(3, 5))
```

1.3 Implement `make_alternator`.

```
def make_alternator(f, g):  
    """  
    >>> a = make_alternator(lambda x: x * x, lambda x: x + 4)  
    >>> a(5)  
    1  
    6  
    9  
    8  
    25  
    """
```

## 2 Lists & Tree Recursion

Mutative (*destructive*) operations change the state of a list by adding, removing, or otherwise modifying the list itself.

- `lst.append(element)`
- `lst.extend(lst)`
- `lst.pop(index)`
- `lst += lst` (**not** `lst = lst + lst`)
- `lst[i] = x`
- `lst[i:j] = lst`

Non-mutative (*non-destructive*) operations include the following.

- `lst + lst`
- `lst * n`
- `lst[i:j]`
- `list(lst)`

*Recall:* To execute assignment statements,

- Evaluate all expressions to the right of the = sign
- Bind all names to the left of the = to those resulting values

The **Golden Rule of Equals** describes how this rule behaves with composite values. *Composite values*, such as functions and lists, are connected by a pointer. When an expression evaluates to a composite value, we are returned the pointer to that value, rather than the value itself.

In an environment diagram, we can summarize this rule with,

Copy *exactly* what is in the box!

2.1 Write a list comprehension that accomplishes each of the following tasks.

- (a) Square all the elements of a given list, `lst`.
- (b) Compute the dot product of two lists `lst1` and `lst2`. *Hint:* The dot product is defined as `lst1[0] · lst2[0] + lst1[1] · lst2[1] + ... + lst1[n] · lst2[n]`. The Python `zip` function may be useful here.
- (c) Return a list of lists such that `lol = [[0], [0, 1], [0, 1, 2], [0, 1, 2, 3], [0, 1, 2, 3, 4]]`.
- (d) Return the same list as above, except now excluding every instance of the number 2: `lol2 = [[0], [0, 1], [0, 1], [0, 1, 3], [0, 1, 3, 4]]`.

2.2 Draw the environment diagram that results from running the following code.

```
pom = [16, 15, 13]
pompom = pom * 2
pompom.append(pom[:])
pom.extend(pompom)
```

2.3 Draw the environment diagram that results from running the following code.

```
bless, up = 3, 5
another = [1, 2, 3, 4]
one = another[1:]

another[bless] = up
another.append(one.remove(2))
another[another[0]] = one
one[another[0]] = another[1]
one = one + [another.pop(3)]
another[1] = one[1][1][0]
one.append([one.pop(1)])
```

2.4 **def** jerry(jerry):  
     **def** jerome(alex):  
         alex.append(jerry[1:])  
         **return** alex  
     **return** jerome

```
ben = ['nice', ['ice']]
jerome = jerry(ben)
alex = jerome(['cream'])
ben[1].append(alex)
ben[1][1][1] = ben
print(ben)
```

## 4 Midterm Review

- 2.5 Implement `subset_sum`, which takes in a list of integers and a number  $k$  and returns whether there is a subset of the list that adds up to  $k$ ? *Hint*: Use the `in` operator to determine if an element belongs to a list.

```
>>> 3 in [1, 2, 3]
True
>>> 4 in [1, 2, 3]
False

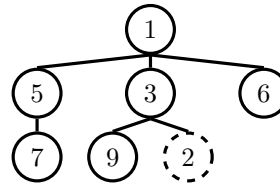
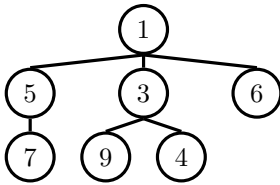
def subset_sum(seq, k):
    """
    >>> subset_sum([2, 4, 7, 3], 5)          # 2 + 3 = 5
    True
    >>> subset_sum([1, 9, 5, 7, 3], 2)
    False
    """
```

- 2.6 Implement `subsets`, which takes in a list of values and an integer  $n$  and returns all subsets of the list of size exactly  $n$  in any order.

```
def subsets(lst, n):
    """
    >>> three_subsets = subsets(list(range(5)), 3)
    >>> for subset in sorted(three_subsets):
    ...     print(subset)
    [0, 1, 2]
    [0, 1, 3]
    [0, 1, 4]
    [0, 2, 3]
    [0, 2, 4]
    [0, 3, 4]
    [1, 2, 3]
    [1, 2, 4]
    [1, 3, 4]
    [2, 3, 4]
    """
```

### 3 Recursion on Trees

- 3.1 A **min-heap** is a tree with the special property that every node's value is less than or equal to the values of all of its branches.



```
def tree(root, branches=[]):
    return [root] + list(branches)
```

```
def root(tree):
    return tree[0]
```

```
def branches(tree):
    return tree[1:]
```

Implement `is_min_heap` which takes in a tree and returns whether the tree satisfies the min-heap property or not.

### 4 Growth

- 4.1 Give a tight asymptotic runtime bound for the following functions in  $\Theta(\cdot)$  notation, or “Infinite” if the program does not terminate.

(a) `def one(n):`  
`while n > 0:`  
`n = n // 2`

(b) `def two(n):`  
`for i in range(n):`  
`for j in range(i):`  
`print(str(i), str(j))`

(c) `def three(n):`  
`i = 1`  
`while i <= n:`  
`for j in range(i):`  
`print(j)`  
`i *= 2`

For each of the questions below, give a  $\Theta(\cdot)$  bound on the asymptotic runtime.

```
4.2 def strange_add(n):
    if n == 0:
        return 1
    else:
        return strange_add(n - 1) + strange_add(n - 1)
```

```
4.3 def belgian_waffle(n):
    i = 0
    total = 0
    while i < n:
        for j in range(n ** 2):
            total += 1
        i += 1
    return total
```

```
4.4 def flip(n):
    return -n
```

```
def pancake(n):
    if n < 1:
        return 1
    return flip(n) + pancake(n - 1) + pancake(n - 1)
```

```
4.5 def toast(n):
    i = 0
    j = 0
    stack = 0
    while i < n:
        stack += pancake(n)
        i += 1
    while j < n:
        stack += 1
        j += 1
    return stack
```